Example Android: Unit Testing

Contents

1 Prerequisites

To use this example you need to install the following tools:

- Android Studio 2.3
- The source and build files of this project :)

---

2 Unit Testing

2.1 Testing

For testing Android apps, you typically create these types of automated unit tests:

- Local tests: Unit tests that run on your local machine only. These tests are compiled to run locally on the Java Virtual Machine (JVM) to minimize execution time. Use this approach to run unit tests that have no dependencies on the Android framework or have dependencies that can be filled by using mock objects.

- Instrumented tests: Unit tests that run on an Android device or emulator. These tests have access to instrumentation information, such as the Context for the app under test. Use this approach to run unit tests that have Android dependencies which cannot be easily filled by using mock objects.

2.2 Preface

Suppose you want to build an Android application in a TDD-style. There are a few nasty things about Android that makes unit testing more difficult than for normal Java applications.

- Android source code is Java but it is compiled by the Dalvik compiler to dex. To run it you need an Android device or emulator.
- In a standard Android Application the Android API is everywhere in your code. The Android code is already tested so to only test *your* application we need to leave Android code outside scope as much as possible. For me *unit testing* means testing *my* code apart from the framework (I consider Android as a library and a framework).

This codebase uses a few libraries to simplify unit testing for Android applications:

- Android Annotations (if you need more power and be able to handle more complex situations consider ButterKnife for UI binding and Dagger 2 for dependency injections, AA does both)
- Robolectric

There are also good design and architecture choices to make that simplify unit testing but this example focuses on a simple application and some libraries. Some resources that might help to improve the design of your Android application are:

- Android Clean Architecture
- Effective Android Architecture

---

3 Android Annotations

3.1 Android Annotations

- AndroidAnnotations is an Open Source annotation processing library that speeds up Android development
- It takes care of the plumbing, and lets you concentrate on what's really important.
- By simplifying your code, it facilitates its maintenance.

3.2 Annotation Processing

- Annotation Processing is a feature of the Java compiler that lets you write a library to process source code annotations at compile time
- Runs your annotation processing code in the compiler and can generate new source code files that will also be compiled
- There is no way with public API to modify existing classes

3.3 How does it work

- AndroidAnnotations is a Java Annotation Processing Tool that generates Android specific source code based on annotations
- It enhances a class by subclassing it with a generated class that adds and overrides methods in the annotated class
- The generated class is named by appending an underscore to the name of the annotated class

---

3.4 How do you use it

Everywhere in the project that you refer to an enhanced class you append an underscore to the class name:

- In the Android Manifest
- In layout resources
- In the code, even in your test code

3.5 Example: Activity and Views

```
@EActivity(R.layout.deckard)
public class DeckardActivity extends Activity {
    @ViewById(R.id.button)
    Button button;
```

```
    @ViewById(R.id.editText)
    TextView editText;

}
```

3.6 Example: Bean and EBean

```
@EActivity(R.layout.deckard)
public class DeckardActivity extends Activity {

    @Bean(ButtonClickListener.class)
    IButtonClickListener buttonClickHandler;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    @AfterViews
    public void afterCreate()
    {
        buttonClickHandler.setActivity(this);
        button.setOnClickListener(buttonClickHandler);
    }
}
```

3.7 Example: Background and UIThread

```
@EBean
class ButtonClickListener implements IButtonClickListener {
    private DeckardActivity activity;
    private IDataController dataController;
```

```
    @Bean(RestDataController.class)
    void setDataController(IDataController dataController){
        this.dataController = dataController;
    }

    @Override
    public void onClick(View view) {
        getAsyncComment();
    }

    @Background
    void getAsyncComment() {
        updateTextField(dataController.getData());
    }

    @UiThread
    void updateTextField(String data) {
        EditText editText = activity.findViewById(R.id.editText);
        editText.setText(data);
    }

    public void setActivity(DeckardActivity activity) {
        this.activity = activity;
    }
}
```

3.8 Android Annotations in Gradle

```
dependencies {
    implementation "org.androidannotations:androidannotations-api:4.4.0"
    annotationProcessor "org.androidannotations:androidannotations:4.4.0"
}
```

---

4 Mocking

4.1 JUnit and Mockito in Gradle (also Powermock for overriding private loggers

Just like in Maven:

```
dependencies {
    testImplementation 'org.mockito:mockito-core:2.3.0'
    testImplementation 'org.powermock:powermock-api-mockito2:1.7.3'
    testImplementation 'org.robolectric:robolectric:3.5.1'
}
```

4.2 Robolectric

Running tests on an Android emulator or device is slow! Building, deploying, and launching the app often takes a minute or more. That's no way to do TDD. There must be a better way.

4.3 Unit testing an Activity

Suppose you want to unit test and activity. You try this:

```
public class DeckardActivityIntegrationTest {

    @Test
    public void whenActivityCreatedAppDependenciesAreSetup() throws Exception {
        DeckardActivity deckardActivity = new DeckardActivity();
        assertNotNull(deckardActivity.buttonClickHandler);
    }
}
```

When you run this test, the console says: "Error: 'Method … not mocked'"

4.4 Error: “Method ... not mocked”

If you run a test that calls an API from the Android SDK that you do not mock, you’ll receive an error that says this method is not mocked. That’s because the android.jar file used to run unit tests does not contain any actual code (those APIs are provided only by the Android system image on a device). Instead, all methods throw exceptions by default, this behaviour can be stopped however:

```
    testOptions {
        unitTests {
            includeAndroidResources = true
            returnDefaultValues = true
        }

    }
```

4.5 Running unit tests without Android device

Robolectric is a unit test framework that de-fangs the Android SDK jar so you can test-drive the development of your Android app. Tests run inside the JVM on your workstation in seconds. With Robolectric and Android Annotations you can write tests like this:

```
@RunWith(RobolectricTestRunner.class)
public class DeckardActivityIntegrationTest {

    @Test
    public void whenActivityCreatedAppDependenciesAreSetup() throws Exception {
        DeckardActivity_ deckardActivity = Robolectric.setupActivity(DeckardActi
        assertNotNull(deckardActivity.buttonClickHandler);
    }
}
```

4.6 Robolectric in Gradle

```
dependencies {
```

```
    testImplementation 'org.robolectric:robolectric:3.5.1'
}
```

Noticed the _ after DeckardActivity? This is the version that comes from Android Annotations (AA) annotation processor. When you use this version all annotations like @Activity and @Bean are resolved. Decide per test cases if you need the real dependencies for @Bean or override the properties with test doubles like stubs or mocks.

4.7 Test Doubles

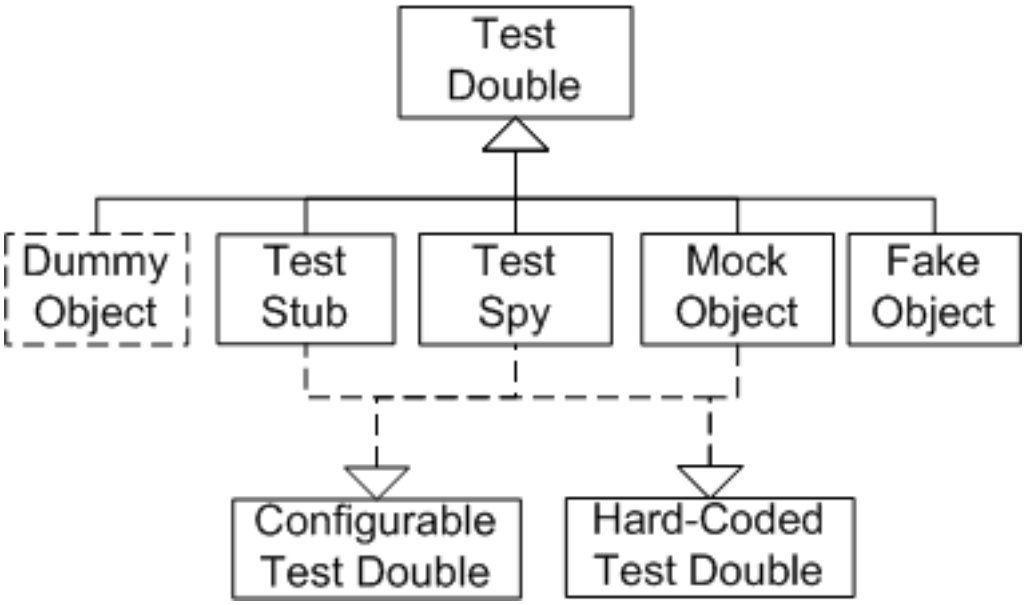

Figure 1: *Figure 1: Different Test Doubles*

---

5 Unit vs Integration Tests

5.1 Unit vs Integration Tests

The DeckardActivityIntegrationTest used the real dependency on RestDataController so it is in fact an integration test. We can also make a unit test for this class:

```
@RunWith(MockitoJUnitRunner.class)
public class DeckardActivityTest {
    @Mock
    Button button;

    @Mock
    ButtonClickListener buttonClickListener;

    @Test
    public void whenActivityCreatedAppDependenciesAreSetup() throws Exception {
        DeckardActivity deckardActivity = new DeckardActivity();
        deckardActivity.button = button;
        deckardActivity.buttonClickHandler = buttonClickListener;
        deckardActivity.onCreate(null);
        deckardActivity.afterCreate();

        verify(buttonClickListener).setActivity(deckardActivity);
        verify(button).setOnClickListener(buttonClickListener);
    }
}
```

Let's look at another class that we can unit test:

```
@EBean
class ButtonClickListener implements IButtonClickListener {
    private DeckardActivity activity;
    private IDataController dataController;

    @Bean(RestDataController.class)
    void setDataController(IDataController dataController){
        this.dataController = dataController;
    }

    @Override
```

```
    public void onClick(View view) {
        getAsyncComment();
    }

    @Background
    void getAsyncComment() {
        updateTextField(dataController.getData());
    }

    @UiThread
    void updateTextField(String data) {
        EditText editText = activity.findViewById(R.id.editText);
        editText.setText(data);
    }

    public void setActivity(DeckardActivity activity) {
        this.activity = activity;
    }
}
```

The ButtonClickListener had two dependencies:

- The DeckardActivity
- The RestDataController

Both dependencies can be mocked:

5.2 Drinking Mockito when you're in AA :)

```
@RunWith(org.mockito.junit.MockitoJUnitRunner.class)
public class ButtonClickListenerTest {
    @Mock
    DeckardActivity deckardActivity;

    @Mock
    IDataController dataController;
```

```
    @InjectMocks
    ButtonClickListener buttonClickListener;

    @Mock
    EditText editText;

    @Mock
    View eventSource;

    @Test
    public void whenButtonClickedTheEditTextContainsHelloOOSE()
    {
        when(deckardActivity.findViewById(R.id.editText)).thenReturn(editText);
        when(dataController.getData()).thenReturn("Hello OOSE");
        buttonClickListener.setActivity(deckardActivity);
        buttonClickListener.setDataController(dataController);
        buttonClickListener.onClick(eventSource);
        verify(editText).setText("Hello OOSE");
    }
}
```

---

6 REST

6.1 Doing REST calls in Android

We saw the RestDataController as a dependency for the ButtonClickListener, but how to do REST calls from an Android device?

There are two things things to consider:

- Android does not allow you to access services on the main thread, luckily AA has @Background and @UIThread (but you can do this with standard Android too)
- You need to alter the AndroidManifest.xml:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

6.2 REST in Android: Retrofit2

There are different options/APIs to do REST calls:

- JAX-RS Client, like Jersey. Heavyweight, has an own DI container (HK2) built on Guice.
- Spring RestTemplate (or use AA's annotation to access this template). Can cause dependency problems with HTTP Commons.
- Use Retrofit2, this is my favourite.

6.3 Retrofit in Gradle

```
dependencies {
    implementation 'com.squareup.retrofit2:retrofit:2.1.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.1.0'
}
```

6.4 Retrofit in your code

With Retrofit the REST interface is mapped to a Java interface like this:

```
interface JsonPlaceHolderAPI {
    @GET("comments/{id}")
    Call<Comment> getComments(@Path("id") int id);
}
```

6.5 Retrofit in your code

Using the Retrofit Builder and a JSON Mapper we can offer a simple Java interface to a client:

```
@EBean
class RetrofitBuilder {
    private final static String BASE_URL = "https://jsonplaceholder.typicode.com/
```

```
    public JsonPlaceHolderAPI create(Class<JsonPlaceHolderAPI> apiClass) {
        Gson gson = new GsonBuilder().setLenient().create();
        return new Retrofit.Builder().baseUrl(BASE_URL).
                addConverterFactory(
                        GsonConverterFactory.create(gson)).build().create(apiCla
    }
}
```

6.6 Retrofit in your code

And the client can use the interface to call methods just like it's a local method invocation:

```
@EBean
public class RestDataController implements IDataController {
    private final static Logger logger = Logger.getLogger(RestDataController.clas

    @Bean
    RetrofitBuilder retrofitBuilder;

    @Override
    public String getData() {
        try {
            return retrofitBuilder.create(JsonPlaceHolderAPI.class).
                    getComments(1).execute().body().getBody();
        } catch (IOException e) {
            logger.log(Level.SEVERE, e.getMessage());
        }
        return null;
    }
}
```

6.7 Integration Testing the RestDataController

To see if the RestDataController really gets the data from our API we can create a simple integration test. We don't need Robolectric here because there are no Android API dependencies to shadow:

```
public class RestDataControllerIntegrationTest {
    @Test
    public void whenFirstCommentIsRequestedItShouldStartWithLaudantium()
    {
        RestDataController_ restDataController = RestDataController_.getInstance
        assertTrue(restDataController.getData().startsWith("laudantium"));
    }
}
```

6.8 Unit Testing the RestDataController

To see if the RestDataController works nicely with the RetrofitBuilder:

```
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

@RunWith(MockitoJUnitRunner.class)
public class RestDataControllerTest {
    public static final String MESSAGE = "message";
    public static final String BODY = "body";

    @Mock
    RetrofitBuilder retrofitBuilder;

    @Mock
    Logger logger;

    @Mock
    JsonPlaceHolderAPI jsonPlaceHolderAPI;

    @Mock
    Call<Comment> commentCall;

    @Test
    public void whenExceptionIsThrownItsMessageGetsLogged() throws IOException {
        RestDataController restDataController = new RestDataController();
        restDataController.retrofitBuilder = retrofitBuilder;
```

```
        when(retrofitBuilder.create(JsonPlaceHolderAPI.class)).thenReturn(jsonPla
        when(jsonPlaceHolderAPI.getComments(1)).thenReturn(commentCall);
        when(commentCall.execute()).thenThrow(new IOException(MESSAGE));
        Whitebox.setInternalState(RestDataController.class, "logger", logger);
```

---

7 Running tests and getting test coverage

7.1 Running tests in Gradle

```
./gradlew test
```

7.2 Getting test coverage for Android projects

Jacoco is a tool for Java application that is able to generate a test coverage file that can be read by SonarQube. We need the following adjustments to build.gradle:

```
apply plugin: 'jacoco'

    buildTypes {
        debug {
            testCoverageEnabled true
        }
    }
```

```
task createTestReports(type: JacocoReport, dependsOn: ['testDevDebugUnitTest', '
    group = "Reporting"

    reports {
        xml.enabled = true
        html.enabled = true
    }
```

```
    def fileFilter = ['**/R.class',
                      '**/R$*.class',
                      '**/BuildConfig.*',
                      '**/*$ViewInjector*.*',
                      '**/*$ViewBinder*.*',
                      '**/*$MembersInjector*.*',
                      '**/Manifest*.*',
                      '**/*Test*.*',
                      'android/**/*.*']
    def debugTree = fileTree(dir: "${buildDir}/intermediates/classes/debug", excl
    def mainSrc = "${project.projectDir}/src/main/java"

    sourceDirectories = files([mainSrc])
    classDirectories = files([debugTree])

    executionData = files("${project.buildDir}/jacoco/testDebugUnitTest.exec")
}
```

8 Resources

- Building Local Tests
- Building Effective Unit Tests
- Unit Test your Application
- Android Annotations Cookbook